Ginger: Implementing a new Lisp family syntax

James Dean Palmer
Northern Arizona University
Flagstaff, Arizona
James.Palmer@nau.edu

ABSTRACT

In this paper we introduce G-expressions, a new syntax based on the S-expression syntax utilized by most Lisp family languages. We have implemented a new homoiconic language, Ginger, based on this syntax and a Smalltalk inspired object system. Like the Scheme language, Ginger employs only a few special forms and observes a minimalist discipline allowing users to define functions that act like the primitive forms in many Algol-like languages. But unlike Scheme, G-expressions allow Ginger to emulate the aesthetic feel of an Algol-like language syntax. While fundamentally a dialect of Lisp, Ginger implements an attractive modern syntax which can superficially resemble Python or Ruby. This syntactic flexibility exemplifies Ginger's true power as a tool for developing task or domain-specific micro-languages.

Categories and Subject Descriptors

D.3.1 [**PROGRAMMING LANGUAGES**]: Formal Definitions and Theory

1. INTRODUCTION

S-expressions are a syntax for representing complex hierarchical data structures. This syntax is most well known for providing the syntactic structure for Lisp, Scheme and other functional languages. S-expressions are composed of symbols, strings, literals, numbers and other S-expressions. Parentheses are used to denote the beginning and end of lists made up of elements separated by white space. Consider this S-expression example:

```
1 (("Lyra" (("Will") ("Roger" (a b)))) (1.73))
```

The expression represents a list with two members. Each member represents another list or S-expression. While essentially hierarchical it can be difficult to determine hierarchical relationships without counting parenthesis or using an editor that supports parenthesis matching.

ACMSE '09 March 19-21, 2009, Clemson, SC

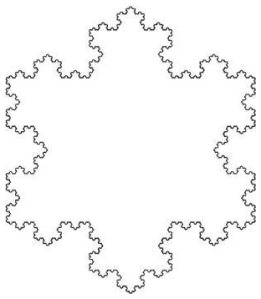

Figure 1: A Koch snowflake is easily constructed from three Koch curves. This figure was generated using Ginger's turtle graphics library which closely mimics Logo's famous turtle.

In this paper we describe *G-expressions*, an S-expressions variant with special rules that incorporate indentation and line continuations to simplify hierarchical notation and improve consistent readable formatting.

An alternate G-expression based rendering of the last example might read:

```
1 "Lyra"
2   "Will"
3   "Roger"
4     a b
5 1.73
```

The horizontal spacing replaces and augments the hierarchical structure provided by S-expression parenthesis. In this example it is much more apparent that `"Lyra"` and `1.73` are at the same hierarchical depth.

G-expressions are not a true superset of S-expressions but the first example happens to also be a valid G-expression. Experience has shown that S-expressions can be difficult for humans to decipher without adding extra formatting and white space to emphasize hierarchical relationships. G-expressions seek to take advantage of the natural organizational spacing most people add to S-expressions. The intent is to reduce parenthesis which are made redundant by explicit formatting choices. While many S-expression and G-expression analogs exist, G-expressions are not simply syntactic sugar for S-expressions. The semantics of structures formed from parenthesis and indentation is different and corresponds to different, but related, internal representations.

This becomes especially significant when G-expressions are used to implement a homoiconic language where the difference effects structure, meaning and interpretation. This has been illustrated in the implementation of Ginger, a new Lisp family language based on G-expressions. The following is an example Ginger program that draws the Koch snowflake in Figure 1 (an equivalent Lisp or Scheme program written using S-expressions is left as an easy exercise for the reader):

```
1 define koch-snowflake (x)
2   repeat 3
3     koch-curve x
4     right-turn 120
5
6 define koch-curve (x)
7   if (< x 5)
8     forward x
9   else:
10    koch-curve (/ x 3)
11    left-turn 60
12    koch-curve (/ x 3)
13    right-turn 120
14    koch-curve (/ x 3)
15    left-turn 60
16    koch-curve (/ x 3)
17
18 koch-snowflake 100
```

2. PREVIOUS WORK

When John McCarthy began implementing Lisp in 1958, the first Lisp programs were written in an informal notation called *M-expressions* which were designed to resemble Fortran. But when the first Lisp interpreter was completed it initially supported only S-expressions - Lisp's list notation. While McCarthy's intent was to eventually implement an M-expressions to S-expression translator it never materialized in part because many programmers preferred the minimalist S-expressions to the more syntax heavy Algol-like syntaxes [4].

In 1997, Ron Rivest sought to formally define S-expressions as a data structure and wrote a draft document describing S-expressions for the Network Working Group [8]. While the document never became an official RFC it is often cited by other RFCs.

S-expressions have remained a popular data format syntax but it is best known for its usage in Lisp family languages. Over the years a number of dialects of Lisp have emerged. Common Lisp and Scheme are easily the most popular dialects in use today but a number of more radical dialects or close cousins have emerged including Logo and Dylan. Both Logo and Dylan are notable for abandoning the S-expressions of Lisp. In Dylan's case, a truly Algol like syntax was adopted (though early versions of Dylan used an S-expression based syntax [1]). The Logo syntax at first glance seems dissimilar from Lisp. But only two major syntactic transformations are necessary to transform Logo code into S-expressions: 1) changing operators from infix to prefix and 2) placing parenthesis at the beginning and end of function calls (which Logo determines by argument count). Many people have noticed Logo's uncanny similarity to Lisp and it is often called "Lisp without the parenthesis."

Several efforts have considered alternate syntaxes for Lisp or Scheme. Most notable are *Sweet-expressions* for Lisp and *I-expressions* for Scheme. Sweet-expressions adds special meaning to grouping symbols (), [] and {}. Curly braces force infix evaluation, parentheses force Algol like function evaluation, hard braces forces an access operator and indentation forces additional parenthesis [12]. Similarly, Scheme SRFI-49 defines I-expressions which uses indentation to directly assert more or less parenthesis in a scheme program's structure [6]. Neither syntax has seen widespread use in part, because one must really understand the underlying language's syntax before superimposing the second syntax. One might also argue these syntaxes seem unnatural with artifacts of the parent syntax embedded as boiler plate code necessary to make the system work. That is, these solutions were developed as transformation functions and not as new forms unto themselves.

3. G-EXPRESSIONS

G-expressions were born in an attempt to make a Logo-like syntax for the Ginger language. Unlike Logo, Ginger supports first class functions that may take optional or even variadic arguments. Thus, Logo's syntax which relies on fixed argument count function calls was unworkable as a mechanism for representing the beginning and ending of expressions. Like Sweet-expressions and I-expressions, G-expressions use indentation to represent hierarchical structures. Each level of indentation represents a special list or grouping of objects. Unlike I-expressions and Sweet-expressions each group represents a single argument to its parent in the hierarchy. Consider the following Sweet-expression:

```
1 define factorial(n)
2   if {n <= 1}
3     1
4     {n * factorial{n - 1}}
```

Lines 3 and 4 in the last example are at the same indentation level but represent separate arguments to the `if` call. This works well enough for a purely functional programming style but begins to look strange and clunky when Scheme style `begin`s or Lisp style `progn`s are introduced:

```
1 define factorial(n)
2   if {n <= 1}
3     progn
4       trace "base case"
5       1
6     progn
7       trace "recursive case"
8       {n * factorial{n - 1}}
```

G-expressions implicitly define `begin`-*blocks* at each new indentation level. The same code written in Ginger would look like this:

```
1 define factorial (n)
2   if (<= n 1)
3     trace "base case"
4     1
5   else:
6     trace "recursive case"
7     * n (factorial (- n 1))
```

Implicit `begin`-blocks help enable imperative programming by allowing sequences of actions to take place without gratuitous `begin`s or abused `let` statements. Another advantage of implicit `begin`-blocks in the context of G-expressions is readability. There is little ambiguity about lines in an indented block representing arguments or serially executed statements. Put another way, explicit `begin`s and similar sequential execution special forms are not needed and Ginger does not implement a `begin` form as part of the language standard.

3.1 Labeled Line Continuations

One might assume that the `if` statement used in the last example is a special syntax form but in fact it follows ordinary function calling conventions. Here the `else:` is a label. When labels begin a line they implicitly continue the previous statement. Labels need not be unique thus allowing this alternative to Lisp and Scheme's `cond`:

```
1 define fib (n)
2   if (= n 0) 0
3   elsif: (= n 1) 1
4   elsif: (= n 2) 1
5   else: (+ (fib (- n 1)) (fib (- n 2)))
```

In this case, the `if` call has 7 arguments and three labels. Consider a lengthier rendering of the same program:

```
1 define fib (n)
2   if (= n 0)
3     0
4   elsif: (= n 1)
5     1
6   elsif: (= n 2)
7     1
8   else:
9     + (fib (- n 1)) (fib (- n 2))
```

When evaluated these two pieces of code give the same output but the expressions themselves correspond to two very different internal representations. The internal representation of G-expressions within Ginger is detailed in section 3.3.

3.2 Anonymous Line Continuations

Without a labeled line continuation, G-expressions normally terminate at the end of a line. But it is often useful and even necessary to span multiple lines without breaking specifically at labels. Ellipsis are used to represent an anonymous line continuation. Consider the next example which represents a long list of arguments to `println`:

```
1 println "a=" a " b=" b " c=" c " d=" d " e="
2 .. e " f= " f " g= " g " h= " h " i= " i " j="
3 .. j " k= " k " l= " l " m= " m " n= " n
```

As with labels, anonymous line continuations can be used as anchors for starting new code blocks:

```
1 println "a="
2   if (null? a) "nothing"
3   else: a
4 .. " b="
5   if (null? b) "nothing"
6   else: b
```

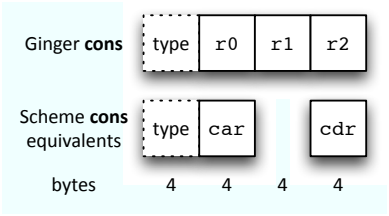

Figure 2: Ginger's cons cell contains three elements. `r0` and `r2` have direct analogs to `car` and `cdr` while `r1` can be used as an extra field which can support the implementation of more advanced data structures.

3.3 Internal Representation

The homoiconic nature of most Lisp family languages creates an elegant mapping between source code and an internal unevaluated representation of the same code. Scheme provides a `read` function that is used to translate raw source into lists and symbols that directly correspond to the underlying S-expressions that make up the program. Since arbitrary Scheme programs can be represented as lists, the `eval` function simply works on these list data structures to execute programs. Yet few implementations of Scheme and Lisp are so pure in their implementation. This simple `read-eval` model ignores important metadata. Line numbers, for example, are extremely useful when evaluation errors occur. Thus, many Scheme implementations either a) adopt much more complicated data structures to represent Scheme program internally or b) incorporate auxiliary data structures to maintain metadata separately from the parsed data. Each approach has its own philosophical drawbacks and diminishes the notion that a clear boundary between `read` and `eval` exists.

Ginger addresses this situation by redefining the underlying representation of lists. Like Lisp and Scheme, Ginger uses `cons` cells to represent lists but Ginger's `cons` cells contain three elements instead of the more typical two elements. Figure 2 describes the mapping between fields `r0`, `r1`, and `r2` in Ginger `cons` cells and `car` and `cdr` in Scheme `cons` cells. One possible structural representation is also described in this figure with each field made up of four bytes along with a 4 byte type field. Ginger uses `r0` and `r2` just as Scheme uses `car` and `cdr` fields to implement lists but additional metadata (like line numbers, column numbers and parse metadata) can be stored in `r1`. While the data in `r1` is not required for correct evaluation, it can provide value-added information and hints for the `eval` function using a well defined metadata representation mechanism that still preserves homoiconicity.

Rich metadata can also improve output from the `display` function which represents internal structures in human readable form. Utilizing embedded metadata, `display` can preserve the exact spacing and formatting used to print a parsed Ginger program exactly as the user has written it. Potentially, comments and editor specific metadata could be embedded in this way to provide more advanced features and analysis.

Dot notation is often used in Lisp and Scheme to represent `cons` cell values when those cells do not form a list. A binary tree with values 1, 2, 3 and 4 at the leaves of the tree could be implemented with `(cons (cons 1 2) (cons 3 4))` and

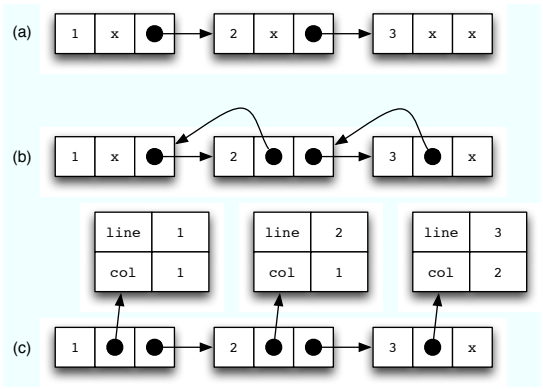

Figure 3: Examples of (a) singly-linked lists (b) doubly-linked lists and (c) metadata preserving lists constructed from `cons` cells.

the resulting tree could be displayed using dot notation as:

```
1 ((1 . 2) . (3 . 4))
```

The analogous expression in Ginger, `(pair (pair 1 2) (pair 3 4))`, would yield:

```
1 ((1 . . 2) . . (3 . . 4))
```

where each center value, `r1`, is `null`. A ternary tree is naturally represented using all three `cons` fields as:

```
1 ((1 . 2 . 3) . (4 . 5 . 6) . (7 . 8 . 9))
```

and other useful data structures such as doubly-linked lists are easily implemented as Figure 3 illustrates.

`begin`-blocks are represented internally as lists but they are formed with a special kind of `cons` cell called a `bcons`. The only difference between a `cons` and a `bcons` is the internal representation of type. This differentiates lists that should be evaluated using normal function call conventions with `begin`-blocks which evaluate each line in the list sequentially. Thus, parenthetically defined lists and indentation defined lists truly represent two different internal representations (and interpretations) and this representation does not depend on the interpretation of the third cons cell element.

3.4 Functions Calls and Evaluation

The Scheme expressions `1`, `(1)` and `((1))` are three very different objects. The same Ginger expressions represent three different objects with a similar in-memory representations as Scheme but the actual evaluation is quite different. In Ginger, every value is an object, every object is a value and every value is a function. `(1)` evaluates to `1` because when values (or objects) are treated as functions with no arguments they simply return their own value. Function calls and variable evaluation are in most circumstances the same thing.

In the following example,

```
1 define one 1
2 define foo ()
3   one
```

there is no ambiguity in the evaluation of `one`. As an integer value it returns its own value if called as a function.

4. MICRO-LANGUAGES AND META-PROGRAMMING

Scheme famously does not include a `while` or `for` loop; instead leaving it as an exercise for the programmer to implement or avoid these iteration based control structures. Scheme boldly omits these forms because its powerful macro system makes implementing these forms efficiently a relatively simple task. Many Lisp family languages have powerful macro systems which allow programmers to write macros in the same language as the underlying program. A few Scheme variants even support first-class macros which support runtime macro expansion and first-class module systems [2]. Several people have suggested such macro facilities make Lisp family languages particularly well suited for implementing micro-languages. A micro-language is a small domain specific language [3]. Micro-languages are closely related to domain-specific languages (DSLs), fourth generation languages (4GLs) and language-oriented programming [9, 5, 11]. One common technique for implementing micro-languages is to base what is meant to be a domain specific language on an existing language which may or may not be domain-specific. But without strong meta-programming capabilities, the ability to realize truly novel domain-specific languages is limited. Lisp and Scheme's strong meta-programming facilities make them well-suited host languages but their S-expressions based syntax may ultimately limit the expressiveness of resulting micro-languages. In the remainder of this section we show several examples of meta-programming and micro-language development accomplished using G-expressions.

4.1 Control Structures

Most Algol-family languages simply don't support the tightly integrated macro system and orthogonal syntax which make meta-programming possible in Scheme. But often the Scheme implementation of an Algol-family control structure doesn't look much like the original. We will show Ginger is well suited for emulating not only control structure functionality but also the aesthetic feel of some very unusual control structures. We accomplish this by way of example and consider several novel control structures used in the Slag language [7]. The Ginger equivalents require no changes be made to the underlying language and simply use G-expressions and macros.

4.1.1 Which (A Multidimensional Switch)

The first Slag control structure we will examine is the `which` statement. A `which` is a multidimensional switch statement:

```
1  which (A,B)
2    case (value1,value1): result = R
3    case (value1,value2): result = S
4    case (value1,value3): result = T
5    case (value2,value1): result = U
6    case (value2,value2): result = V
7    case (value2,value3): result = W
8    case (value3,value1): result = X
9    case (value3,value2): result = Y
10   case (value3,value3): result = Z
11 endWhich
```

In this example, `result` would equal `R` if `(A == value1)` and `(B == value1)`. Similar multi-variable checks happen at each line until no more cases exist. The same structure can be implemented with a single macro in Ginger that provides a similar feel:

```
1 which (A B)
2 case: (value1 value1) (set result R)
3 case: (value1 value2) (set result T)
4 case: (value1 value3) (set result T)
5 case: (value2 value1) (set result U)
6 case: (value2 value2) (set result V)
7 case: (value2 value3) (set result W)
8 case: (value3 value1) (set result X)
9 case: (value3 value2) (set result Y)
10 case: (value3 value3) (set result Z)
```

4.1.2 Contingencies

The second Slag control structure we will examine is the `contingent` form. This form executes a block of code until a contingency is either satisfied by meeting sufficient or necessary conditions or fails by failing to meet the necessary conditions. This control structure is fairly sophisticated because the keywords `necessary` and `sufficient` can appear anywhere in the code as illustrated in this example:

```
1 local Int32[] primes()
2 forEach (n in 101..201 step 2)
3   contingent
4     necessary (n % 2 != 0)
5     forEach (d in 3..sqrt(n) step 2)
6       necessary (n%d != 0)
7     endForEach
8   satisfied
9     primes.add( n )
10  endContingent
11 endForEach
```

The same control structure can be implemented efficiently in Ginger by defining a `contingency` macro which takes two required unlabeled arguments and two optional labeled arguments (`satisfied:` and `unsatisfied:`), a `necessary` macro which takes two required unlabeled arguments and a `sufficient` macro which takes two required unlabeled arguments. The same example written in Ginger becomes:

```
1 define primes '()
2 each n (range 101 201 step: 2)
3   contingency valid-prime
4     necessary valid-prime (!= (% n 2) 0)
5       each d (range 3 (sqrt n) step: 2)
6         necessary valid-prime (!= (% n d) 0)
7   satisfied:
8     set primes (pair n primes)
```

In general, Ginger adopts a Scheme-like minimalist philosophy supporting very few special forms and intrinsics but we feel powerful macro facilities make almost any kind of control structure possible. This certainly seems to be the case for Scheme where most Scheme Requests for Implementations (SRFIs) contain sample implementations for functionality that often add radically new functionality to the language.

4.2 OpenGL and State Machines

Many useful libraries are at their core state machines. OpenGL is an excellent example and often requires the developer to enable and disable modes. But there is little reason much of this state book keeping couldn't be done in the language itself. One typical pattern for drawing in OpenGL is:

```
1 glBegin(GL_POLYGON);
2 glVertex3d(a,b,c);
3 ...
4 glEnd();
```

Should `glBegin()`s and `glEnd()`s not properly begin and terminate in order, the programs behavior is undefined and may even crash. This can easily be addressed in Ginger by making `glBegin` and `glEnd` part of a macro which accomplishes both. The equivalent Ginger code would be:

```
1 glBegin 'gl-polygon
2   glVertex a b c
```

where `glBegin` is a macro that calls `glBegin()` with the first macro argument, executes the second macro argument and then calls `glEnd()`. Similar macros could be devised for `glPush()`/`glPop()` pairs:

```
1 initialize-matrix
2 glPush
3   manipulate-matrix
4   glPush
5     manipulate-matrix
6     glPush
7       manipulate-matrix
8
9 print-matrix
```

Here, `print-matrix` would print the initialized matrix as the effects of each manipulate-matrix get popped off the matrix stack on line 8 as each hierarchical block ends.

4.3 Simulating Infix Evaluation

We end this section by considering infix evaluation which is not disallowed by G-expressions (or S-expressions) but first-class support is for various reasons uncommon in many Lisp family languages. A novel consequence of making Ginger's variable evaluation essentially the same thing as function evaluation is that variables can in essence take parameters when being evaluated. Consider this simple example:

```
1 define a 7
2 define b 3
3 a + b
```

We can read line 3 as saying, "evaluate `a` with argument `+` and `b`". If `a` is treated like a functor, then we can assign a function that performs this evaluation which applies infix evaluation to the remaining arguments. While this neatly integrates infix evaluation it does so using dynamic dispatch (`a`'s type must be determined before the correct functor definition can be found) and this is not as efficient as calling `+` as a prefix operator (which may in fact be implemented as a macro or as a call to an intrinsic function).

5. FORMAL SYNTAX DEFINITION

Finally, we give the BNF syntax for G-expressions as described in this paper:

```
<gexprs>    ::= <gexpr>+
<gexpr>     ::= <bgn-line> (<block>){0,1}
                (<cnt-line> <block>{0,1})*
<block>     ::= <indent> <gexprs> <dedent>
<bgn-line>  ::= (<token> | <list>) <items> <eol>
<cnt-line>  ::= (<label> | <ell>) <items> <eol>
<list>      ::= "(" <white-space>* (<items> |
                <null>) <white-space>* ")"
<items>     ::= <item> | <item> <white-space>+
                <items> <white-space>*
<item>      ::= <token> | <list> | <label>
<label>     ::= <token> ":"
<token>     ::= <token-char1>* <token-char2>
<string>   ::= "\"" (<token-char2> |
                <white-space> | <eol> |
                (<escape-char> "\""))* "\""
<token-char1> ::= <token-char2> | "." | ":"
<token-char2> ::= <alpha-char> | <digit-char>
                  | <symbol-char>
<symbol-char> ::= "!" | "@" | "#" | "$" | "%" |
                  "^" | "&" | "*" | "_" | "-" |
                  "+" | "=" | "|" | "~" | "/" |
                  "<" | ">" | "?"
<escape-char> ::= "\"
<alpha-char>  ::= "a" | ... | "z" |
                  "A" | ... | "Z"
<digit-char>  ::= "0" | ... | "9"
<white-space> ::= " " | "\t"
<ell>       ::= ".."
<eol>       ::= "\n"
<null>      ::= ""
<indent>    -- a relative increase in the amount
               of leading whitespace; leading
               whitespace is otherwise ignored.
<dedent>    -- a relative decrease in the amount
               of leading whitespace; a dedent
               must decrease to a previously
               defined indentation level
```

The <indent> and <dedent> tokens were inspired by Python's lexical description. Implementation details for <indent> and <dedent> are described in The Python Language Reference [10]. We also note that Ginger's syntax is slightly more involved as it supports more complicated quoting and escaping rules beyond the scope of this paper.

6. CONCLUSION

In this paper we have presented a new syntax for data representation. G-expressions intend to form a sound syntactic basis for a new multi-paradigm language we are developing called Ginger. We have given a formal definition of this syntax, suggested an internal organization that preserves source file metadata and provided examples of the expressiveness and power of both Ginger and G-expressions. We believe Ginger is an ideal host language for meta-programming, domain specific programming and developing micro-languages. Ginger has the soul of a Lisp but the aesthetic of Python.

7. REFERENCES

[1] Apple Computer Eastern Research and Technology. *Dylan(TM) an object-oriented dynamic language*. 1992.

[2] A. Bawden. First-class macros have types. In *In 27th ACM Symposium on Principles of Programming Languages (POPL'00*, pages 133–141. ACM, 2000.

[3] J. Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, 1986.

[4] J. McCarthy. History of LISP. *History of programming languages I*, pages 173–185, 1981.

[5] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[6] E. Möller. SRFI 49: Indentation-sensitive syntax. `http://srfi.schemers.org/srfi-49/srfi-49.html`, 2003.

[7] A. Pralle. Slag language primer. `http://www.plasmaworks.com/files/slag/slag_primer.html`, 2008.

[8] R. L. Rivest. Network working group internet draft: S-expressions. `http://people.csail.mit.edu/rivest/Sexp.txt`, 1997.

[9] D. Spinellis. Notable design patterns for domain-specific languages. *J. Syst. Softw.*, 56(1):91–99, 2001.

[10] G. Van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., September 2003.

[11] M. P. Ward. Language-oriented programming. *Software — Concepts and Tools*, 15(4):147–161, 1994.

[12] D. Wheeler. Sweet-expressions: A suite of readable formats for Lisp-like languages. `http://www.dwheeler.com/readable/sweet-expressions.html`, 2006.